COMP3700 Machine Learning

Lecture 3&4 The backpropagation algorithm

We have so far investigated two learning algorithms, namely the perceptron convergence procedure and the delta rule. Both of these algorithms make use of errors measured at individual neural elements in order to determine required adjustments to thresholds and weights. It was pointed out that when a network contains hidden units, these algorithms cannot be applied because any error associated with a hidden unit is unknown (there is no target value against which to compare its output). It was stated toward the end of the last lecture that the algorithm known as backpropagation was devised to overcome this problem and that this algorithm requires that the hard-limiting step function of the LTU be replaced by a sigmoid. We will now derive the backpropagation algorithm in terms of the network shown in figure 3.1.

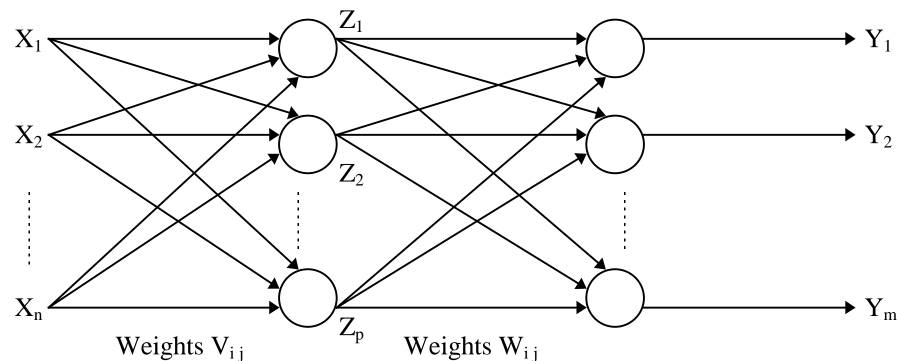


Figure 3.1 A two-layer feedforward network

The above network has two layers of variable weights and we refer to such a network as a two-layer feedforward network. Clearly the network has one layer of hidden units. In the *input layer*, weight V_{ij} connects input X_j to hidden unit i, whose output is Z_i. In the *output layer*, weight W_{ij} connects hidden unit j to output unit i whose output is Y_i.

Backpropagation, like the earlier learning rules we have examined, has been developed for use in the case of *supervised learning*, ie the case where, for any given input vector **X**, the required output is known. We will call this required output the target vector **T** and define the error function

$$E = \frac{1}{2}\sum_{i=1}^{m}(Y_i - T_i)^2 \qquad (3.1)$$

This function computes half the sum of the squares of the differences between the network outputs and the target outputs. The half is included because it slightly simplifies the derivation of the algorithm.

It is usual to commence the learning process with the weights set to small random values. With this as a starting point we can assume that the error function is non-zero. The objective of a learning algorithm is to reduce the error function and the starting point for developing the backpropagation algorithm is the observation that we can reduce the error function by a small amount if we make a small change ΔW_{ij} in any of the W_{ij} such that

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} \qquad (3.2)$$

where η is a small positive constant.
The weight Wij connects hidden unit j to output neuron i so the total input to neuron i (ie its *activation*) is

$$I_i = \sum_{j=1}^{p} W_{ij} Z_j \qquad (3.3)$$

In order to calculate ΔW_{ij} from equation (3.2), we need an expression for $\partial E/\partial W_{ij}$. Note that

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial I_i} \cdot \frac{\partial I_i}{\partial W_{ij}} = \frac{\partial E}{\partial I_i} . Z_j \qquad (3.4)$$

and we can also write

$$\frac{\partial E}{\partial I_i} = \frac{\partial E}{\partial Y_i} \cdot \frac{\partial Y_i}{\partial I_i} = \frac{\partial E}{\partial Y_i} . F'(I_i) \qquad (3.5)$$

where $Y_i = F(I_i)$ is the input-output law of the neural processor. Note that this is differentiable if the characteristic is sigmoidal, but is not differentiable if the neuron has a hard-limiting characteristic as in the LTU.

Differentiating (3.1) gives

$$\frac{\partial E}{\partial Y_i} = Y_i - T_i \qquad (3.6)$$

so that, from (3.4), (3.5) and (3.6)

$$\frac{\partial E}{\partial W_{ij}} = Z_j (Y_i - T_i) F'(I_i) \qquad (3.7)$$

A sigmoidal expression for F(.) that is widely used is

$$Y_i = \frac{1}{1 + e^{-I_i}} \qquad (3.8)$$

which is depicted in Figure 3.2.

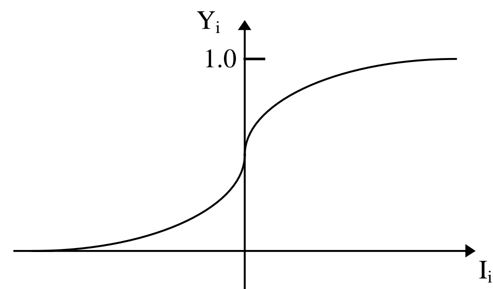


Figure 3.2 The shape of equation (3.8)

With this expression for F(.) we have

$$F'(I_i) = \frac{\partial Y_i}{\partial I_i} = Y_i (1 - Y_i) \qquad (3.9)$$

Substituting this into (3.7) allows us to write, for the required change to W_{ij}:

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} = -\eta Z_j (Y_i - T_i) Y_i (1 - Y_i) \qquad (3.10)$$

Thus, when an output pattern is compared to a target pattern, each connection to output processor i should have its weight adjusted according to (3.10).

This has shown how to adjust the weights in the W_{ij} layer in the network in figure 3.1. We now follow a similar procedure to determine how the weights in the V_{ij} layer should be adjusted.
Let J_j represent the activation of hidden unit j. J_j can then be written

$$J_j = \sum_k V_{jk} X_k \qquad (3.11)$$

and, if the hidden units have characteristic G(.), we can write

$$Z_j = G(J_j) = G(\sum_k V_{jk} X_k) \qquad (3.12)$$

Output unit i therefore receives the input

$$I_i = \sum_j W_{ij} Z_j = \sum_j W_{ij} G(\sum_k V_{jk} X_k) \qquad (3.13)$$

and produces at its output

$$Y_i = F(I_i) = F(\sum_j W_{ij} G(\sum_k V_{jk} X_k)) \qquad (3.14)$$

A small reduction in the error function in equation (3.1) can be brought about by changing the input weights such that ΔV_{jk} is proportional to $-\partial E/\partial V_{jk}$ and, in order to evaluate this derivative, it is clear that we will need to use the chain rule a few times because V_{jk} is fairly deeply nested in equation (3.14). The error function is

$$E = \frac{1}{2} \sum_i \left[T_i - F(\sum_j W_{ij} G(\sum_k V_{jk} X_k)) \right]^2 \qquad (3.15)$$

where the summation over i is a sum over *m* terms. Each of these *m* terms is of the form below, where the outputs and activations of network units have been marked explicitly to assist with the application of the chain rule.

$$E_i = \frac{1}{2}\Big[T_i - F\Big(\sum_j W_{ij} G(\sum_k V_{jk} X_k)\Big)\Big]^2 \qquad (3.16)$$

J_j

Z_j

I_i

Y_i

Before we apply the chain rule, note that

$$E = \sum_i E_i \qquad \text{and} \qquad \partial E / \partial V_{jk} = \sum_i \partial E_i / \partial V_{jk}$$

and to determine the latter derivative, which is our main objective, we apply the chain rule to equation (3.16) going from the inner terms to the outer terms as follows:

$$\frac{\partial E_i}{\partial V_{jk}} = \frac{\partial E_i}{\partial J_j} \cdot \frac{\partial J_j}{\partial V_{jk}} = \frac{\partial E_i}{\partial J_j} . X_k \qquad (3.17)$$

$$\frac{\partial E_i}{\partial J_j} = \frac{\partial E_i}{\partial Z_j} \cdot \frac{\partial Z_j}{\partial J_j} = \frac{\partial E_i}{\partial Z_j} G'(J_j) \qquad (3.18)$$

$$\frac{\partial E_i}{\partial Z_j} = \frac{\partial E_i}{\partial I_i} \cdot \frac{\partial I_i}{\partial Z_j} = \frac{\partial E_i}{\partial I_i} . W_{ij} \qquad (3.19)$$

$$\frac{\partial E_i}{\partial I_i} = \frac{\partial E_i}{\partial Y_i} \cdot \frac{\partial Y_i}{\partial I_i} = \frac{\partial E_i}{\partial Y_i} F'(I_i) \qquad (3.20)$$

$$\frac{\partial E_i}{\partial Y_i} = Y_i - T_i \qquad (3.21)$$

Substituting (3.21) → (3.20) → (3.19) → (3.18) → (3.17) gives

$$\frac{\partial E_i}{\partial V_{jk}} = (Y_i - T_i) F'(I_i) W_{ij} G'(J_j) X_k \qquad (3.22)$$

and hence

$$\frac{\partial E}{\partial V_{jk}} = G'(J_j) X_k \sum_i (Y_i - T_i) F'(I_i) W_{ij} \qquad (3.23)$$

and if the neurons in both layers have the sigmoidal characteristic defined in equation (3.8),

$$\text{ie} \qquad F(I_i) = \frac{1}{1 + e^{-I_i}} \qquad \text{and} \qquad G(J_j) = \frac{1}{1 + e^{-J_j}}$$

then

$$\frac{\partial E}{\partial V_{jk}} = Z_j (1 - Z_j) X_k \sum_i (Y_i - T_i) Y_i (1 - Y_i) W_{ij} \qquad (3.24)$$

and the required change to V_{jk} is simply given by

$$\Delta V_{jk} = -\eta \frac{\partial E}{\partial V_{jk}} \qquad (3.25)$$

And so, for the network in figure 3.1, we have established the manner in which the weights in the two layers of the network should be changed in order to reduce (slightly) the output error defined by equation (3.1). These required changes, defined by equations (3.10) and (3.25), are often abbreviated as follows

$$\Delta W_{ij} = \eta \delta_i Z_j \quad \text{where} \quad \delta_i = -(Y_i - T_i) F'(I_i) \qquad (3.26)$$

and $$\Delta V_{jk} = \eta \delta_j X_k \text{ where } \delta_j = G'(J_j)\sum_i \delta_i W_{ij} \quad (3.27)$$

where the derivative of F is given by equation (3.9) and the derivative of G is given by an equation of the same form.

The above weight-update equations were derived for the case where the network is required to learn a single input/output mapping. That is, the case where the network is to learn to produce a particular output vector **T** when a particular input vector **X** is applied at the input terminals. In this case, if vector **X** is repeatedly applied and the changes defined by equations (3.10) and (3.25) are repeatedly made, the error will eventually be reduced to zero.

For most applications, however, we are not interested in the situation where a single input/output mapping is to be learnt. Usually, we want to train a network to give correct output vectors for many different input vectors. For this more general, and more typical, case, the derivation of the weight update equations proceeds very similarly to the above. The only significant change is that the network error has to be measured over the full set of input vectors, so the error function takes the form:

$$E = \frac{1}{2}\sum_{p,i}(Y_i^p - T_i^p)^2 \quad (3.28)$$

In this equation we not only sum the squared error over the output terminals (the sum over i), we also sum over all input vectors (the sum over p). The symbol p is used here for the input vectors because it is common to refer to these vectors as patterns. I will very often use this terminology in the future.

Having defined the error function (3.28) we can now follow exactly the same derivation procedure as before, using the chain rule, to determine the changes that should be made to the network weights in order to cause a (small) reduction in the value of the error function. These required changes are given by:

$$\Delta W_{ij} = \eta \sum_p \delta_i^p Z_j^p \text{ where } \delta_i^p = -(Y_i^p - T_i^p)F'(I_i^p) \quad (3.29)$$

and $$\Delta V_{jk} = \eta \sum_p \delta_j^p X_k^p \text{ where } \delta_j^p = -G'(J_j^p)\sum_i \delta_i^p W_{ij}^p \quad (3.30)$$

Comparing these equations with (3.26) and (3.27) shows that the weight update formulae in the case where multiple mappings have to be learnt differ from those in the single mapping case only in that the effects of all patterns have to be taken into account, and this is done by the summations over p.

Note that equations (3.29) and (3.30) have the same form, differing only in the definition of the δs. If we were to extend our analysis to networks with more than two layers of variable weights, we would find that the weight update formulae maintained this particular form. That is, for a feedforward network with an arbitrary number of layers, the backpropagation update rule always has the form

$$\text{Weight change} = \eta \sum_p \delta_{output} \times V_{input} \quad (3.31)$$

where the subscripts *output* and *input* refer to the terminals of the weight to be changed. V_{input} is simply the value of the signal appearing at the input terminal of the weight. The meaning of δ_{output} depends upon the network layer that the weight belongs to. For the output layer (W_{ij} layer), δ is given by the RHS of equation (3.29), but for all other layers, δ is given by an equation like the RHS of (3.30). This is easily shown by applying the chain rule to feedforward networks with more than two layers.
Implementation of the weight update rules (3.29) and (3.30) involves applying the full set of input patterns to the network and observing the error in each case before making any adjustments to the weight values. (Only after the full set of errors has been observed can equations (3.29) and (3.30) be evaluated.) Proceeding in this way ensures accurate calculation of the error gradient and that the weight updates follow the direction of steepest descent in the error function. When backpropagation is implemented in this way it is said to be operating in *batch mode* because the full batch of input patterns is presented to the network before any weight updates are carried out. The fact that batch mode follows the direction of steepest descent in the error function guarantees that it will find a set of weights that correspond to a minimum of the error function. But batch mode is, in fact, less frequently used than *incremental mode*, in which the weights are updated after the presentation of each pattern.

One of the disadvantages of batch mode is that implementation of (3.30) and (3.31) requires additional storage of information for each weight during presentation of the patterns. Thus, incremental mode is simpler to implement and, because weight changes are carried out after the presentation of each pattern, steepest descent is not followed and instead, a degree of randomness is introduced into the descent procedure. It is widely believed that this randomness assists in avoiding *local minima*. (A local minimum of the error function is a minimum that gives a relatively poor solution - usually there are many minima in a function to be minimised and only one, the *global minimum*, corresponds to the smallest possible value of the function.) Thus, it is believed that the randomness introduced by the incremental backpropagation procedure gives a better chance of finding the global minimum of the error function. Certainly the incremental approach has been found to be superior for most applications.

The incremental algorithm is summarised as a step-by-step procedure in the following books:
"Introduction to the Theory of Neural Computation" by J Hertz, A Krogh and R G Palmer, Addison-Wesley, 1991, pages 119-120.
"Neural Networks, a Comprehensive Foundation" by S Haykin, Macmillan, 1994, pages 155-156.
A pseudocode version of the incremental algorithm appears in the review paper:
"Progress in Supervised Neural Networks" by D R Hush and B G Horne, IEEE Signal Processing Magazine, January 1993, page 14.
A public domain version of the algorithm will be made available to you for practice in its use in a tutorial.

Note that the backpropagation procedure involves the forward propagation of signals generated by a given input pattern, through the network, to the output. The error is then calculated and the δs, which provide a measure of the contribution to the error from each weight, are then calculated for each layer, starting with the output layer and moving backwards towards the input. In other words, the procedure involves backpropagation of errors, and hence its name. One significant feature of the backpropagation algorithm is that the update rule is *local* to the weight being updated. That is, in order to compute a weight change, the only quantities needed are the strength of the signal at the weight input and the δ that is propagated back from the next layer forward. Importantly, this means that the computations can be carried out in parallel, one layer at a time. But perhaps more interestingly, learning in biological systems must (if you think about it) take place using local information. Does this mean that the backpropagation algorithm is similar to learning procedures employed by the brain? Not really. The fact that backpropagation involves forward and backward transmission of signals through network links indicates a need for bidirectional connections which do not exist in biological systems. And although backpropagation can be organised so as to operate with unidirectional links, neuroscientists have found no evidence to support the idea that backpropagation might be used by the brain.

Applications of backpropagation

The backpropagation algorithm has been applied to a very wide range of problems. We will here examine several applications starting with a couple of "toy" problems that are sometimes used as examples when coding up a new algorithm or when comparing the performance of different learning algorithms.

Two "toy" problems

(i) Exclusive OR

We have already seen one solution to the XOR problem. This solution was discussed in lecture 2 and is reproduced in Figure 3.3. The structure in Figure 3.3 is not a conventional multilayer feedforward network, but such structures are trainable by variants of the backpropagation algorithm, so long as the neural units have sigmoidal characteristics. In the structure depicted here, the units are LTUs, with their threshold values (0.5 and 1.5) written in as shown. As pointed out in lecture 2, the hidden unit in this network computes the logical AND of the input variables and this, coupled with the negative weight, allows it to inhibit the output unit when both inputs have the value unity.

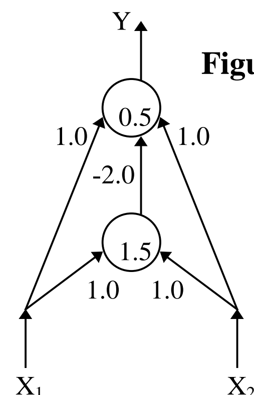


Figure 3.3

A solution to the XOR problem using a more conventional feedforward structure (still using LTUs) is shown in figure 3.4. In this network, the left-hand unit computes the logical OR and the right- hand unit computes the logical AND. As with the previous network, the AND unit inhibits the output when both inputs have the value unity.

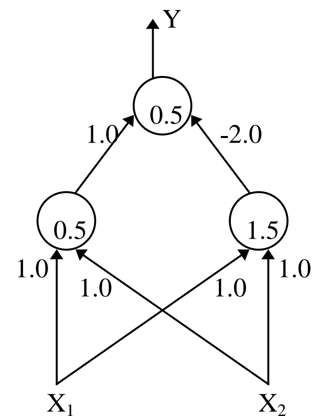


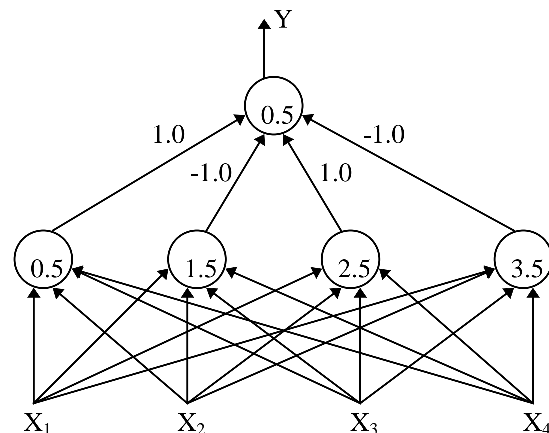
Figure 3.4

Now, it is obvious that we can take network structures such as those in figures 3.3 and 3.4 and simply use trial and error to find a set of weights that solves the XOR problem. The weights shown on the two figures have, in fact been found by this means. But suitable weights can also be found by backpropagation so long as the neural units have sigmoidal characteristics.

There are problems, however, in applying backpropagation, and we will frequently encounter these problems as this course proceeds. One problem is that backpropagation is very slow. Even for small networks like these, if you commence with random valued weights, it takes hundreds of repeats of the steps of the algorithm (on the average) in order to obtain a solution. Another problem is that the weight values tend to become much larger than those shown in the figures. This is largely because the backpropagation algorithm attempts to drive the sigmoidal units into saturation (in order to achieve outputs close to 0 or 1).

(ii) Parity

The parity problem is essentially a generalisation of the XOR problem to N inputs. The single output is required to be on if an odd number of inputs is on, and should be off otherwise. A solution for the case of four inputs is shown on Figure 3.5, where the units are LTUs. In this figure, given that all the input weights are equal to unity, it is clear that hidden unit j will be on if j or more inputs are on. The alternating nature of positive and negative weights in the output layer then guarantees that the network operates as required (as you should check).



Note: All input weights have value unity

Figure 3.5 A solution to the 4-input parity problem

As with the networks above that implement XOR, the network here constitutes a contrived solution to the problem. Backpropagation could be used to train a network with this structure to find a solution to the parity problem so long as units with sigmoidal characteristics were employed. But, as before, when backpropagation is employed on a problem such as this, the weights obtained tend to have quite large values.

You should appreciate that the parity problem constitutes an extreme form of learning problem because the output is required to change whenever a single input changes value. This makes the solution quite hard to learn. Obviously the problem is not typical of most real-world classification problems, which generally have much greater regularity - inputs that are close together usually belong to the same class and hence require the same output.

More practical problems

In the two problems considered above, the training set normally includes all possible input patterns so that the network is never required to provide a response to patterns that it has not been exposed to during training. In other words, the network is not required to *generalize* upon the behaviour it has learnt during training. In the remaining problems that we will consider, the training set forms only part of the problem domain, and the network is expected to generalize on previously unseen cases.

NETtalk

The NETtalk project was carried out a little over ten years ago and was aimed at training a neural network to pronounce English text. The basic structure employed is illustrated in figure 3.6 which indicates that the input consists of seven consecutive characters from written text, presented in a moving window which gradually scans the text. The output units drive a speech synthesizer and so have to be trained to produce an output vector that causes the synthesizer to pronounce correctly the letter in the centre of the window. Note that a window of reasonable length is necessary here because the manner in which a letter is pronounced can be very much influenced by letters on either side. A width of seven seems to be sufficient for all but a few exceptionally difficult cases.

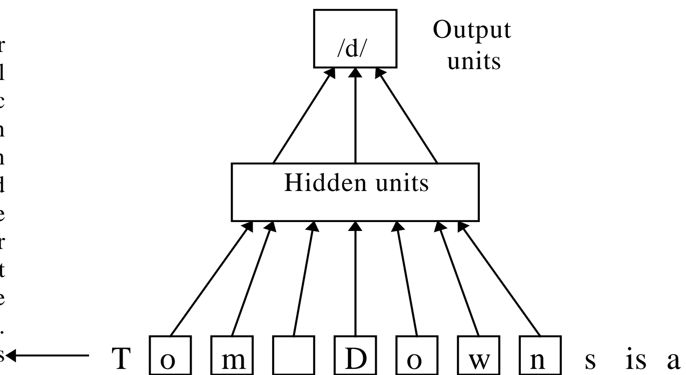


Figure 3.6 Basic NETtalk structure

In the input layer, each of the seven "blocks" in the window has to be able to respond appropriately to each of the 26 alphabetic characters, as well as to some of the symbols used for punctuation. Accordingly, each of the seven blocks in the input window contains 29 neural units and, during operation, just one unit in each block fires, indicating which of the alphabetic characters, space, full stop or comma, is present at each block. Thus, there are 203 input units in total.

The hidden layer contains 80 units each of which receives an input from all 203 units in the input layer. And the output of each hidden unit is fed to each unit in the output layer (of which there are 26). The 26 output units allow representation of 23 articulatory features with the other 3 units being used to represent stress. This arrangement gives the output units sufficient power to elicit from the speech synthesizer most of the basic sounds that we produce in constructing our speech. These basic sounds are known as *phonemes*, and there are about 50 phonemes in English speech (the exact figure varies according to which speech scientist you talk to). In this system, the speech synthesizer is induced to produce individual phonemes by the firing of different combinations of output units. Note that the NETtalk architecture is a *fully-connected* feedforward structure (every unit in the input layer is connected to every unit in the hidden layer, and every unit in the hidden layer is connected to every unit in the output layer) and , given the number of units in each layer, this implies that the number of connections (ie weights) in the network is nearly 20,000.

Initial training of the network was carried out using samples of children's speech available from the literature. This was followed by further training using words selected from a dictionary. After roughly 12 hours of cpu time on a VAX 780 (a leading minicomputer of the day, which had less power than a present-day desktop machine) the system was correctly reproducing phonemes from the training set around 95% of the time. A demonstration of NETtalk's training performance will be given to you in class.

After training, the network was tried out on previously unseen text and produced quite intelligible speech, although its accuracy in terms of correct phoneme production slipped to 78%. This is an illustration of a very important capability of artificial neural networks. After training they can be exposed to previously unseen inputs and will usually perform acceptably well. When we measure the performance of a neural network on data that was not used in training (such data is usually called the *test data*) we are measuring the neural network's ability to *generalize* on what it learnt in training. The fact that neural networks usually exhibit good generalization performance is a major reason why they have become the focus of much attention in recent years. The performance results obtained using NETtalk are typical in the sense that generalization performance was inferior to performance obtained on the training data (you would suspect that something was wrong if this did not occur). But for most applications a generalization performance of 78% would be rather inadequate and additional training (probably on a broader training set) would normally be required.

Investigations were also carried out to see how gracefully performance of the system would degrade if the network was subjected to damage. In a series of experiments, random noise values were added to the weights and some neural units were removed. Graceful degradation in performance was observed. The fact that performance degradation is usually graceful in artificial neural networks (as in their biological counterparts - see lecture 1) is

another reason why neural computing has emerged in recent years as a subject of considerable interest among engineers and scientists. The reason why performance degradation is usually graceful stems from the manner in which learnt knowledge in a neural network is *distributed* over the network rather than being stored at specific sites as in conventional computer systems. When conventional computer systems are subjected to damage in the way that the NETtalk system was, performance degradation can be expected to be catastrophic rather than graceful.

Before we move on to the next practical application of neural networks, it is worth comparing NETtalk with a system known as DEC-talk which had similar objectives to NETtalk but was developed by DEC using a conventional computing approach. The conventional approach to developing a system that pronounces English text involves attempting to establish the linguistic rules that govern the manner in which we as humans convert the text we read into speech sounds. This is an enormous task and DEC employed a team of linguists who spent ten years drawing up a very large and comprehensive set of rules. As a result of all this work, DEC obtained a system that is undoubtedly superior to NETtalk in performance. This superiority, however, is not that great and, most importantly, NETtalk simply learnt from a set of examples - there was no need for an army of experts to produce a complex set of rules. This is a major advantage of artificial neural networks (ANNs). They are easy to construct and can be used to solve problems that are difficult to define mathematically (eg the mapping of text to speech). It should be added, however, that in cases where a given problem is well enough understood, conventional systems, based upon algorithms or rules, usually perform better than ANNs. This is the case with the text-to-speech problem where ten years of study by experts has led to the problem being sufficiently well understood for the conventional approach to provide a superior system.

A paper that provides additional details on the NETtalk system is: T J Sejnowski and C R Rosenberg, “ Parallel networks that learn to pronounce English text”, Complex systems, Vol 1, pp145-168, 1987.

Sonar target recognition

This application is also due to Sejnowski and co-workers. (Terrence Sejnowski has joint appointments at the Salk Institute and the University of California, San Diego and his work in the mid-1980’s did much to stimulate the current world-wide interest in artificial neural networks.) The main paper in which the sonar target recognition work is described is: R P Gorman and T J Sejnowski, “Analysis of hidden units in a layered network trained to classify sonar targets”, Neural Networks, Vol 1, pp75-89, 1988.

The problem looked at in this paper essentially concerns attempting to distinguish between rocks and mines in the sea by means of sonar. This is, at least conceptually, a much simpler problem than the conversion of text to speech, because there is much less variability in the signals. Nevertheless, although the return signals that have bounced off the object (either rock or mine) contain information about the shape of the object (either irregular rock or smooth mine) it would be very difficult, if not impossible, to implement the deconvolution process necessary to obtain the shape of the object from the sonar return. And so, in existing systems used for tasks of this type, the (high frequency) sonar return is shifted in frequency to the audio range so that humans can be trained to carry out the required classification.

In class you will have an opportunity to listen to the audio version of these sonar returns and to try to distinguish between the two types. Untrained listeners apparently tend to do little better than pure chance on this task but sufficient training can bring them up to a classification accuracy of about 90%.

Gorman and Sejnowski trained a neural net with a similar structure to the one used for NETtalk to distinguish between rocks and metal cylinders lying at the bottom of Chesapeake Bay. The sonar signal employed was a “chirp” pulse , ie a signal whose frequency increases linearly over the pulse duration. In an attempt to reveal to the neural network some of the structure in the return signal, Gorman and Sejnowski subjected the signal to Fourier analysis to produce a time-frequency plot of the form shown in figure 3.7(b). (Note how the return signal maintains the basic form of a chirp, with its frequency components increasing linearly with time.) A set of 60 frequency slices of the signal were then taken using a set of sampling apertures of the form shown in figure 3.7(a). Effectively, these apertures were superimposed over the time-frequency plot, as shown in figure 3.7(b) and then the amount of power in each frequency slice was calculated by integrating over each aperture. This integration provided the power-spectral density plot shown in figure 3.7(c).

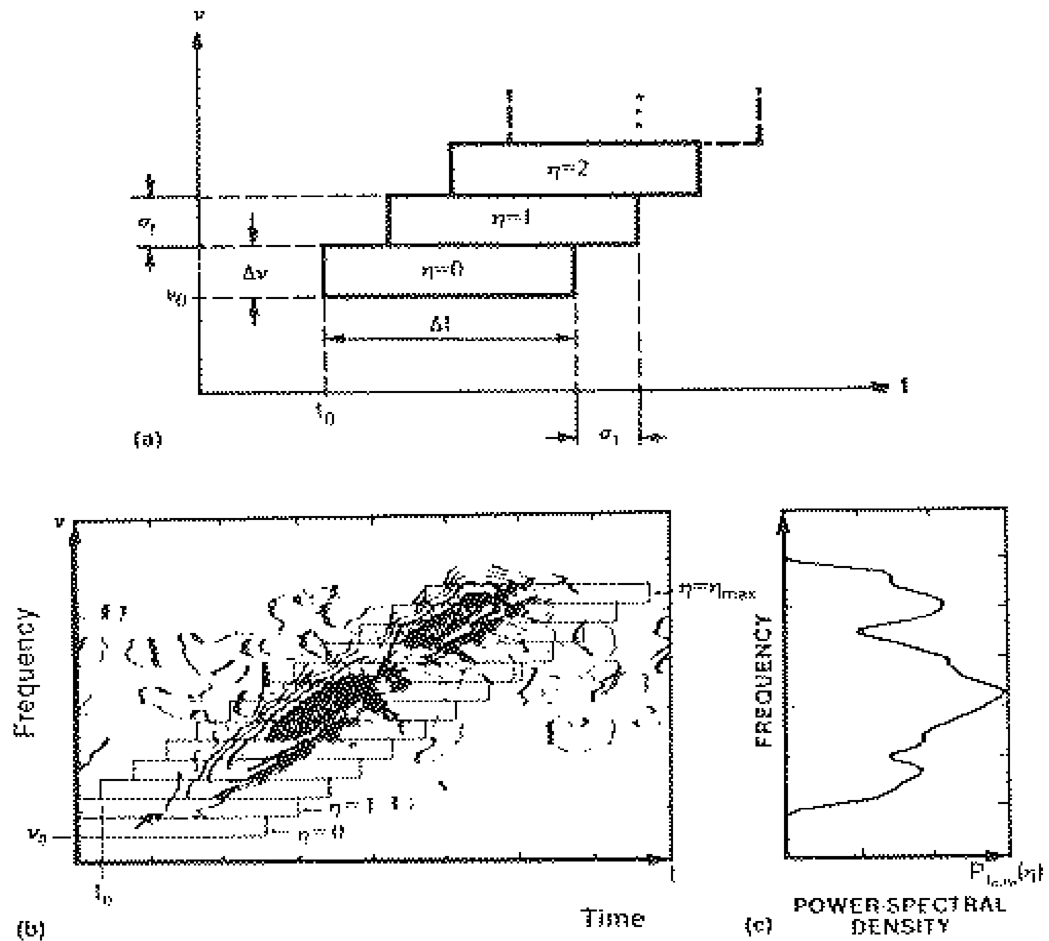


Figure 3.7 Preprocessing the sonar return to obtain the power-spectral density

Note that the plot of power-spectral density is not continuous as shown in figure 3.7 but consists instead simply of 60 values obtained by integrating over the 60 frequency bands. Having obtained the power-spectral density in this form, Gorman and Sejnowski then fed its 60 values to the input units of their neural network, as illustrated in figure 3.8.

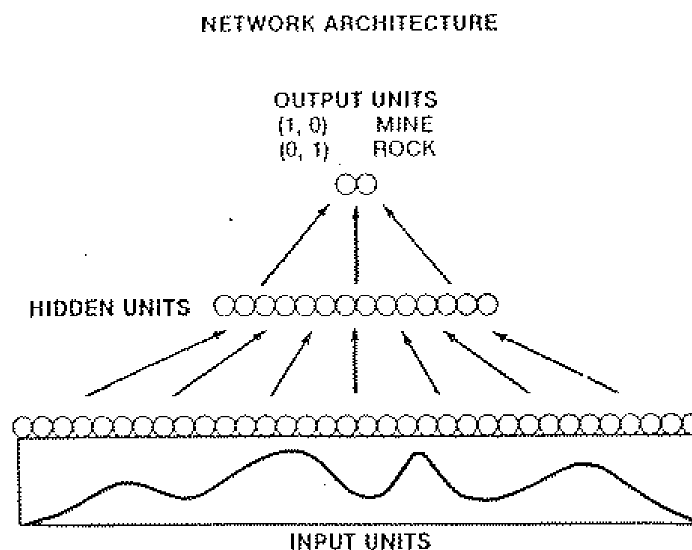


Figure 3.8 The neural network with power-spectral density input

The network was trained using backpropagation on 200 sonar returns, 100 of each type. Note that figure 3.8 indicates that Gorman and Sejnowski chose to use a network with two outputs when a single output would have sufficed. They apparently did this because they believed that this would give them a better chance of establishing a set of rules that described the manner in which the network solved the problem. As far as I know, they never managed to find the rules although the network did manage to solve the problem quite successfully with training taking about 2 hours on something slower than a present-day desktop machine.

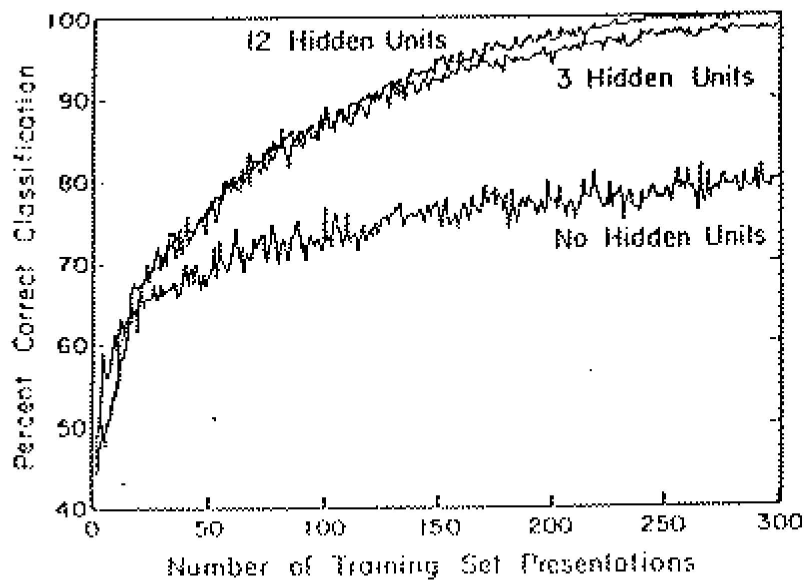


Figure 3.9 Performance obtained from networks with different numbers of hidden units

Figure 3.9 illustrates the kind of learning performance achieved by neural networks with zero, three and twelve hidden units. Table 1 shows that with twelve hidden units, performance on the training set reached 99.8% accuracy and that 100% accuracy was achieved with 24 hidden units. But, of course, performance on the training set is not the best guide to how good a tool has been produced by the training. Information processing tools are developed for application on previously unseen data so that performance on a set of data that were not used in training (the test set) is a much better guide. The fact that the network with 24 hidden units managed 100% accuracy on the training set (in which noise was undoubtedly present) is a good indication that this particular network has over-fitted itself to the data - it has learnt extra features that belong to the noise rather than the sonar returns. And this can be seen in Table 1 where the performance of the network with 24 hidden units on a set of returns not used in training was inferior to the performance obtained from a network with 12 hidden units.

Number of hidden units	Performance on training set (%)	Performance on test set (%)
0	79.3	73.1
2	96.2	85.7
3	98.1	87.6
6	99.4	89.3
12	99.8	90.4
24	100.0	89.2

Table 1

Table 1 shows that the best performance achieved on the test set was 90.4% correct classification. It is interesting to compare this performance with that attainable using more conventional classification methods. Probably the

simplest method is the *nearest-neighbour rule* which stores the full set of training examples and compares any new example (ie from the test set) with the training set. This comparison allows determination of the training example which is closest to the new example and the new example is then given the same classification as that training example. Gorman and Sejnowski applied the nearest-neighbour rule to the sonar data and obtained a performance on the test set of 82%.

Having obtained this result, we can now predict the very best classification performance that could be obtained for this problem. For any statistical classification problem the very best possible classifier is known as the *Bayes optimal classifier* and an important theorem (published in the paper below) states that the optimum Bayes classifier would give performance half way between nearest-neighbour performance and 100% accuracy. So if we assume that the 82% achieved by the nearest-neighbour classifier is a good estimate of the performance it would give for a much larger set of examples, then the theorem tells us that the very best that any classifier could achieve on this problem is 91% accuracy. Thus the 90.4% achieved by the network with 12 hidden units is very close to the best that could be achieved.

The paper in which the above theorem appeared is "Nearest neighbour pattern classification" by T M Cover and P E Hart, IEEE Transactions on Information Theory, Vol IT-13, pp21-27, 1967.

The fact that the best result obtained for the sonar return classifier was determined by experimentation with the number of hidden units indicates the existence of an important problem: how do we decide how many hidden units we should use. Unfortunately, experimentation is currently the best way and, of course, such experimentation is quite tedious. Techniques have been developed that allow you to commence with a network that you know is larger than you need and then "prune" it after training. Other techniques start off with a network with a single hidden unit and introduce additional hidden units as necessary during training in order to accommodate the problem. Neither of these approaches is particularly satisfactory and, if time permits, we will investigate their weaknesses later.

Navigation of a car

D A Pomerleau, "ALVINN: An autonomous land vehicle in a neural network," in Advances in Neural Information Processing Systems, Vol 1, D S Touretsky (ed), pp 305-313, 1989.

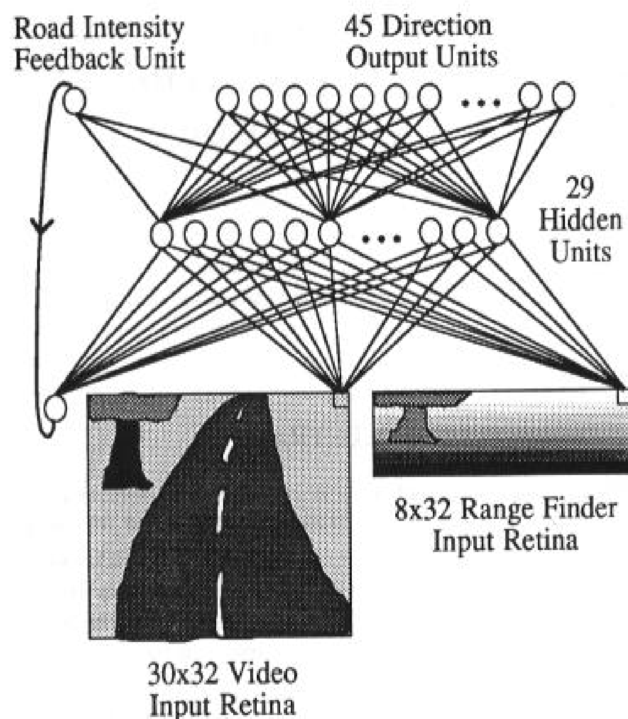


Figure 3.10

Pomerleau's system is a neural net controller for driving a car on a winding road. Autonomous systems for vehicle control had been developed previously using more conventional computer vision and robotic techniques but they all suffered from deficiencies, particularly in the face of the noise and variability associated with real world scenes. The neural architecture employed in Pomerleau's system was a feedforward network with a single hidden layer. The input layer was fed with visual information of three types, as shown in figure 3.10. Pomerleau referred to two of these input types as "retinas". The larger of the two retinas fed information to the network from a 30 x 32 pixel image of the road ahead obtained from a TV camera mounted on the roof of the vehicle. Each of the 960 units receiving this information was fed an input whose value was proportional to the intensity in the blue colour band of the image. (The blue region was apparently chosen because it gave the best contrast between road and non-road.) The smaller retina, consisting of 8 x 32 units, received its input from a laser range finder and the activation in each unit of this retina was proportional to the proximity of the corresponding area in the image. The one remaining input (the road intensity

feedback unit) simply gives an indication of whether the road in the image currently being analysed is lighter or darker than the non-road in the previous image. Each of the total of 1217 input units is fully connected to the hidden layer of 29 units which, in turn, is fully connected to the output layer.

The 45 output units indicate the direction that the vehicle should follow. The network was trained so that only one output unit should be on at any time and if the unit in the centre of the 45 output units was ON, this indicated that the vehicle should proceed straight ahead. If a unit to the left (right) was ON, this indicated that the vehicle should turn to the left (right), the degree of turn depending on just how far to the left or right the ON unit was. The objective was always to aim to bring the vehicle to the centre of the road seven metres ahead of the current position.

The network was trained on 1200 simulated road images using backpropagation. After 40 epochs of training on each of the 1200 images the network performed well enough for the vehicle to drive through a wooded area near the Carnegie-Mellon (CMU) campus at speeds of about 10Km/hr. This is obviously not too impressive, but in later work (see the book below which is based upon Pomerlau's PhD thesis) Pomerlau removed the range finder, which was largely there for object avoidance, and a correspondingly smaller ANN was able to drive the vehicle on a highway at the US speed limit of 55 mph (nearly 90 Km/hr). The vehicle had to be driven in the lane furthest from exit ramps to avoid confusing the network. Its record distance at full speed was 21.2 miles (34 Km).

It is worth noting that in his paper of 1989, Pomerleau points out that once the training scheme had been set up, backpropagation was able to produce in half an hour a road-following system whose performance could only be matched by the vision and autonomous navigation groups at CMU after several months of algorithm development and parameter tuning.

The book referred to above (which is in the PSE library) is: D A Pomerlau, "Neural network perception for mobile robot guidance", Kluwer, 1993.

Image compression

In order to conserve bandwidth, it is important, especially in applications such as high-definition TV, to take advantage, as much as possible, of the considerable redundancy that normally exists in the images that one wishes to transmit. In the case of television there is generally enormous redundancy from frame to frame, except when there is a sudden change of scene. In addition, there is also considerable redundancy from line to line in an image. Image compression methods are designed to take advantage of redundancies such as these so that images can be transmitted at much lower bit rates than would be necessary without compression.

Compression methods have also been developed over the years for images other than those arising in TV, for example, to reduce bandwidth requirements for images transmitted back from deep space. Thirty years ago, techniques were reported for compressing images sent back from moon landings. These techniques were based upon classical transform methods such as the two-dimensional Fourier transform. An example is shown in figure 3.11(b) which is the Fourier transform of the image (from the Surveyor moon landing craft) in figure 3.11(a). Bandwidth reduction is possible using the Fourier transform because the image energy, which is normally fairly uniformly distributed in the spatial domain, tends to be concentrated near the origin in the Fourier domain. This indicates that most of the high-frequency components of the spatial image are of very low magnitude and need not be transmitted. And so, in this example, only the brighter-looking parts of the Fourier domain image need be transmitted. The image is reconstructed at the receiving end by inverting the transform. The result of this operation is shown in figure 3.11(c) and it can be seen that although some of the high frequency information has been lost, the basic image remains. These images come from a paper which also demonstrates how another type of transform, the Hadamard transform, tends to concentrate all the energy of an image in the bottom left hand corner and actually gives better results than those attainable by the Fourier transform. The paper is: W K Pratt, J Kane and H C Andrews, "Hadamard transform image coding," Proceedings IEEE, Vol 57, pp 58- 68, 1969.

The success of these transform techniques indicates that there is a considerable degree of redundancy in nearly all images, so it is of interest to investigate whether ANNs can achieve similar, or even better, image compression results. The problem can be formulated as a supervised learning problem by training a feedforward network to reproduce at its output an image that is presented at its input. Compression is achieved by having a smaller number of hidden units than there are input units. This is illustrated in simplified form in figure 3.12. Here the network would be trained on an eight pixel image presented at its inputs, the objective being to reproduce the

image as well as possible at the output terminals. When training is complete, presentation of the image at the input terminals leads to a compressed version of the image appearing at the outputs of the hidden units. This compressed version can be transmitted and, when fed through the second layer of weights at the receiver, the image is restored. An experiment in which this approach was employed is described in:
G W Cottrell et al., "Learning internal representations from gray-scale images," Proc 9th Annual Conf of Cognitive Science Society, pp 462-473, 1987.

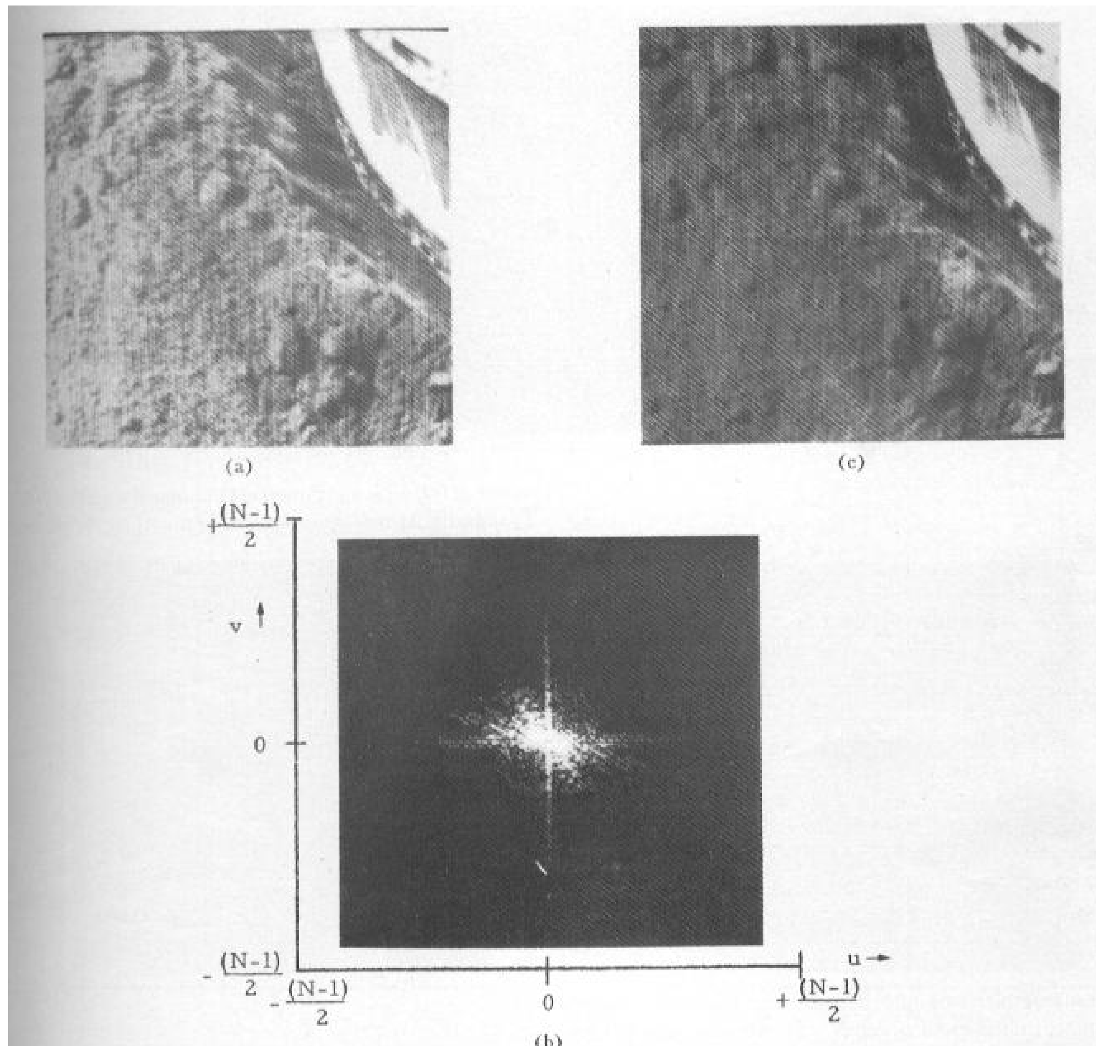


Figure 3.11

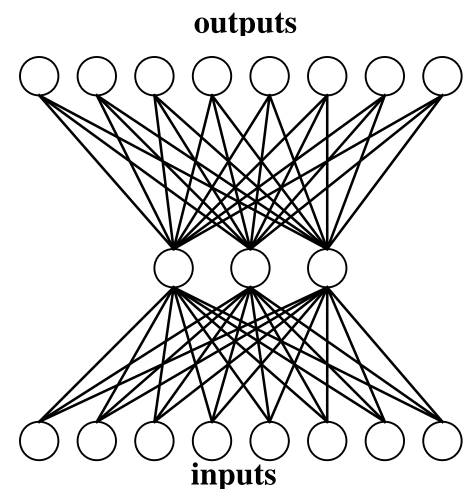


Figure 3.12

The network studied in this paper took its input from 8x8 pixel regions of an image (64 input units, each employing 8 bit gray-scale precision) and had 16 hidden units. Thus, the degree of compression was quite high. The network was trained by standard backpropagation on randomly selected patches of a particular image for typically 150,000 training steps. Then it was tested on the entire image, patch by patch, using a complete set of nonoverlapping patches. Near state-of the-art results were obtained using this simple procedure.

As one might expect, performance was worse when the network was tested on very different pictures, but it was still very respectable.

Recognition of handwritten digits

The technique described in this section is drawn from the following paper:

Y Le Cun et al., "Handwritten digit recognition with a backpropagation network," Advances in Neural Information Processing Systems, Vol 2, pp 396-404, 1990.

The aim of this project was to develop a system for zipcode (postal code) recognition that could be employed in mail sorting in the United States and the database they used consisted of more than 9000 numerals from mail passing through the Buffalo post office. In a sense, this is a fairly straightforward learning task for ANNs because there are only ten different digits that have to be recognised, but a major problem stems from the fact that the digits are handwritten and therefore subject to enormous variability. Some examples that bear this out are shown in figure 3.13. The digits were written by many different people, using a great variety of numeral sizes, writing styles and instruments and with widely varying levels of care. The database was supplemented by a set of more than 3000 printed digits coming from 35 different fonts. The training set consisted of a little over 7000 handwritten digits and 2500 printed digits. The remaining 2000 handwritten and 700 printed digits were used as the test set. The printed fonts used in the test set were different from the printed fonts used in the training set.

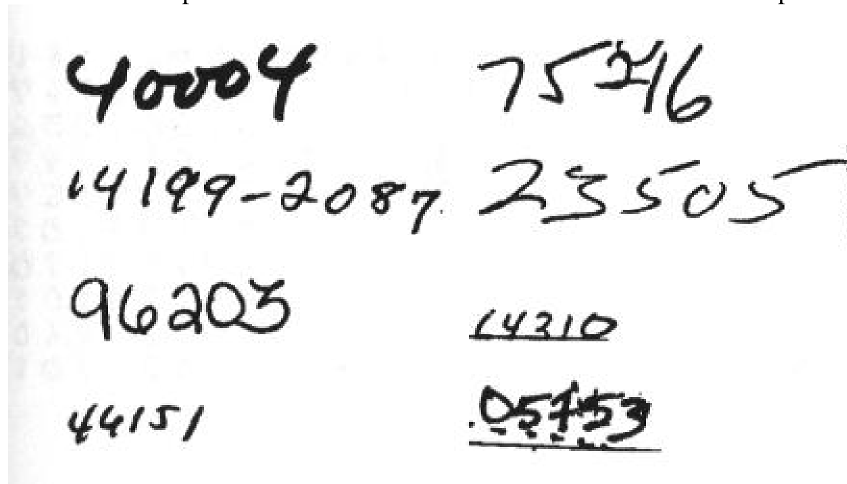


Figure 3.13

In order to render the learning task reasonably tractable, a substantial amount of preprocessing of the digits was carried out using conventional AI techniques before the digits were fed to the ANN. This included in particular the issues of location of the zipcode on the mail item and digit segmentation. The latter involves separation of each digit from its neighbours and this is a difficult task given the fact that digits are often run together or themselves not fully connected (especially the 5's) - see figure 3.13. But once these tasks have been carried out the digits are available as individual items but are still generally of different sizes. So, a normalisation step is also carried out in order to render all digits equal in size. A set of normalised, segmented digits is shown in figure 3.14.

Figure 3.14

The digits are now ready to be fed to the ANN which is a feedforward network with three hidden layers. The input is a 16x16 array that corresponds to the size of a normalised pixel image, as illustrated in figure 3.15. The

first hidden layer contains 12 groups of units with 64 units per group. Each unit in a group is connected to a 5x5 square in the input array and all 64 units in a group have the same 25 weight values. This is known as *weight sharing* and not only reduces training time but also ensures that each unit in a particular group responds to the same feature at different places in the input array. Thus the hidden units in the first hidden layer are known as feature detectors and indeed, the second hidden layer, which consists of 12 groups of 16 units, operates in a similar fashion, but now seeks features in the first hidden layer. The third hidden layer consists of 30 units that are fully connected to the units in the previous layer and the output units are in turn fully connected to the units in the third hidden layer.

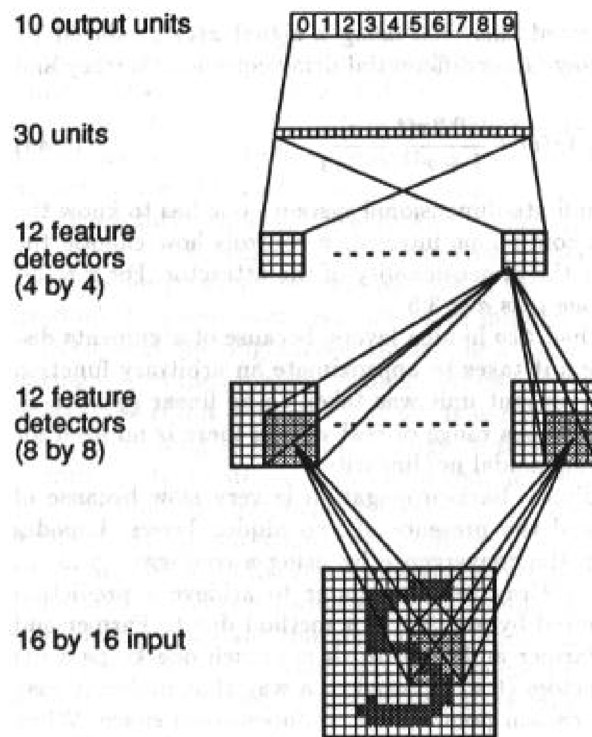


Figure 3.15

Using the training and test sets described above an error rate of about 1% was achieved on the training set and an error rate of 5% on the test set. The error on the test set could be reduced to about 1% by rejecting marginal cases where there was insufficient difference between the strongest and next strongest outputs. These results are close to what is achievable by a human classifier (and better than those achievable by a *tired* human classifier).

Note that a three-layer network was employed in this application. Why three layers? There is no simple answer generally to the question of how to choose an architecture for a particular application. In this particular case, the choice was probably influenced by the fact that the human visual system is layered in a similar fashion to this network. We will return to the issue of architecture selection later.

Interconnection network control

This is a more recent application which is reported in the paper below which won the prize for best paper at the conference at which it was presented. (M F Sakr, C L Giles et al., "On-line prediction of multiprocessor memory access patterns," Proceedings IEEE International Conference on Neural Networks, 1996, pp 1564-1569).The second author, Lee Giles, who is based at the NEC Research Institute in Princeton, New Jersey, was in our department and gave a seminar on this work in late 1996.

Figure 3.16 depicts a shared memory mutiprocessor system which contains an interconnection network which responds to requests from the processors P_i for access to the memory modules M_j. There will, of course, be contention between requests - only one processor can access a given memory module at any one time and also the interconnection network (IN) has a limited number of communication channels. Consequently there is a need for an IN controller and the above paper presented the performance of an ANN-assisted controller.

A straightforward way of organising this kind of system is to make it *demand-driven* which means that each time a processor needs to access a particular memory module it makes a request to the IN controller to establish a suitable path. When such a system is executing a particular application, the memory-access requests made by the processors tend to follow a pattern that arises from the application. The goal of an ANN would then be to predict as well as possible the requests that the various processors will make so that the controller can then configure the IN as efficiently as possible. To achieve this the ANN must learn the processor-memory access patterns and predict as well as possible any changes that take place in these patterns.

The system depicted in figure 3.16 is not demand driven but instead uses a technique known as state sequence routing (SSR). In SSR the existence of patterns in the memory requests is assumed and the controller takes the network through a fixed set of configurations in a repetitive manner. The sequence of configurations is held in a shift register called the state generator and the state generator broadcasts this sequence repetitively to each of the

processors, memory modules and switching elements. A processor needing to access a particular memory module then examines the broadcast sequence and only needs to issue a request to the controller if a suitable access path

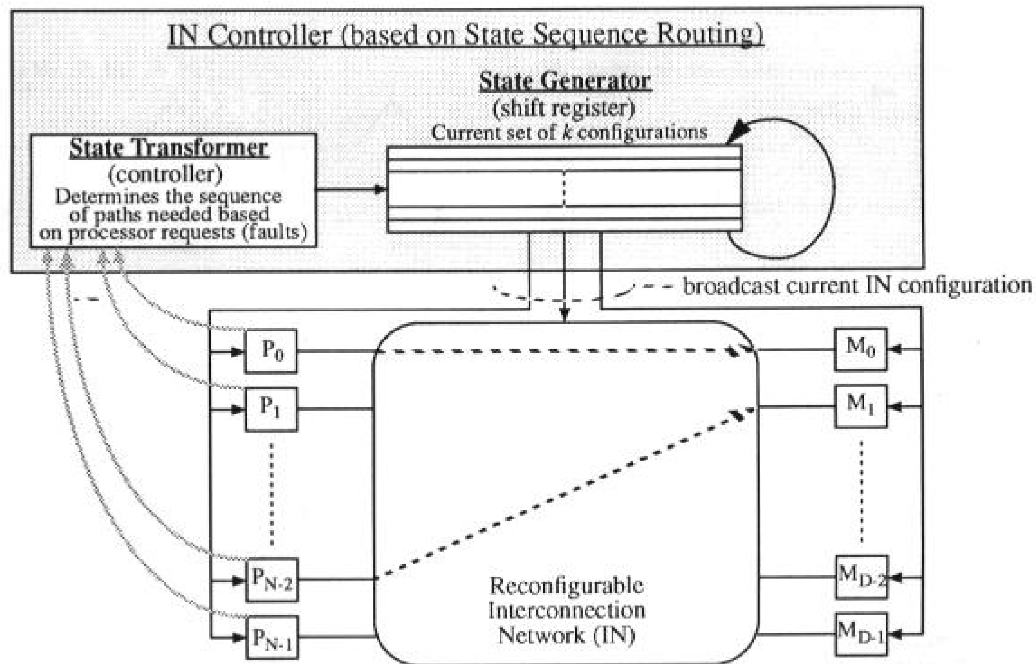


Figure 3.16

does not appear in the sequence. In response to such a request (which is called a *fault*) the controller will replace an existing configuration in the state sequence by one that includes the required path.

In this scenario the goal of the ANN is to reduce the likelihood of faults occurring by predicting changes in memory access patterns and informing the controller of needed changes before faults occur. In this way the controller can transform the state sequence so as to include soon-to-be-needed paths.

Since the processor-memory access patterns change dynamically, they can be modelled as a time series and well-established techniques of time series analysis could be used to make the required predictions. But these well-established methods would require a lengthy history of memory access behaviour in order to make good predictions. So, in this study, a type of ANN that can make short-term predictions, called a time delay neural network (TDNN), was employed. A simple example of such a network is shown in figure 3.17. The network is a feedforward structure of the same kind as we have previously met and is trainable by backpropagation in the usual way. The only difference from the earlier networks is that the inputs to the network in figure 3.17 are values of one variable at successive time instants, ie x(t), x(t-τ), x(t-2τ), etc. Backpropagation could be employed to train this TDNN to predict future values of the variable x(.).A training set would consist of a reasonably lengthy series of values of the variable x(t), call them x(0), x(1), x(2), etc. Training would commence with the first five values of this series, ie x(0) to x(4) as inputs and the network would try to predict the value of x(5). Next, the network would be fed x(1) to x(5) as inputs and it would try to predict the value of x(6). Training would continue in this way until the full set of data had been used and then the process would be repeated as many times as necessary to obtain satisfactory predictions.

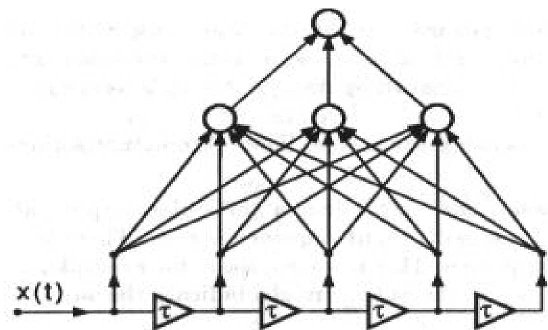


Figure 3.17 A time delay neural network

For the interconnection network controller, a separate TDNN was trained for each processor. Each TDNN was

a little more complicated than the one depicted in figure 3.17. Instead of a scalar input like x(t), a vector input, which we'll call **x**(t), was necessary. The vector **x**(t) had 8 entries, corresponding to the 8 memory modules in the multiprocessor system, and contained a 1 in position j (and zeros everywhere else) if the processor accessed module j at time t. The other vector inputs **x**(t-τ), **x**(t-2τ), etc. were similarly arranged. Each TDNN had 8 outputs, and during training, output i was given the value 1 (and all others were given the value zero) if the next module accessed by the processor (after the sequence of accesses presented at the input) was module i. In this way the TDNN was trained to predict future memory module accesses requested by the processor. Each TDNN had 10 hidden units.

Examples of the improvements obtained (in terms of the reduction in the number of faults) are shown in figures 3.18 and figure 3.19.. Figure 3.18 shows performance on a partial differential equation solver and figure 3.19 shows performance on a program for implementation of the fast Fourier transform. The latter has more regular module accesses and so better performance is obtained, but note how in each case, it takes the neural network time to learn the pattern of access.

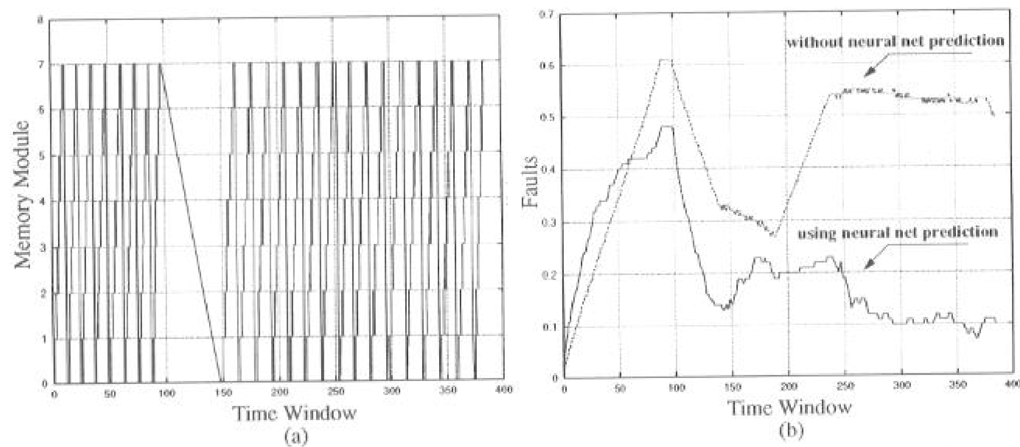


Figure 3.18

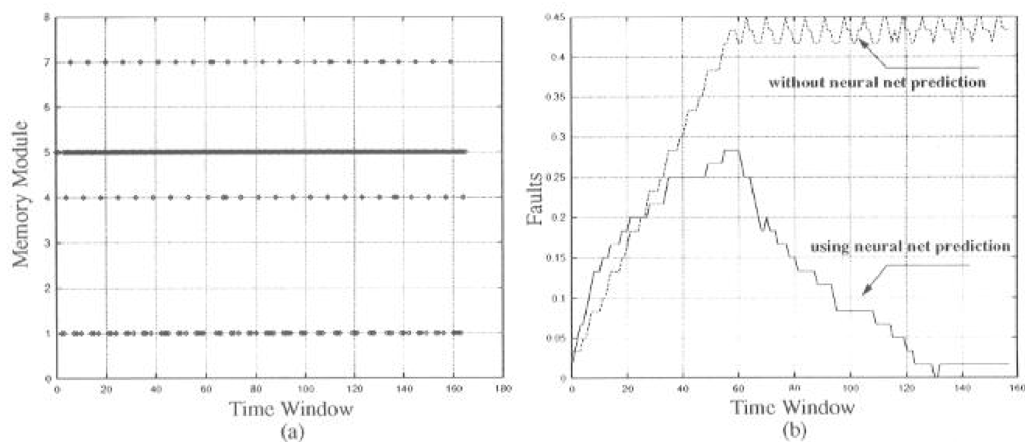


Figure 3.19